

# Practical State Machines for Computer Software and Engineering

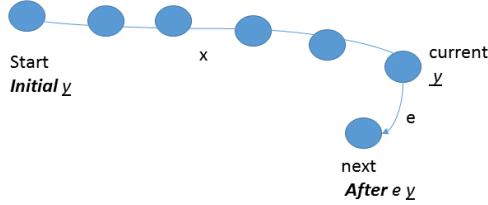
Victor Yodaiken

February 23, 2016

## Abstract

This paper introduces methods for describing properties of possibly very large state machines in terms of solutions to recursive functions and applies those methods to computer systems.

## 1 Introduction



A computer program or computing device changes state in discrete steps in response to discrete events. If  $E$  is the set of events and  $E^*$  is the set of finite sequences over  $E$ , then each element  $s \in E^*$  describes a sequence of events that drives a system from an initial state to some *current state*. An equation of the form

$$\underline{y} = F(x)$$

where  $x$  is a free variable over  $E^*$  defines  $\underline{y}$  as a discrete “state variable”. The underline is a mental aid to remind us that state variables of this sort depend on the sequence parameter. If  $\underline{y} = F(x)$  and  $\underline{z} = G(x)$  then

$$\underline{y} < \underline{z}$$

is true if and only if  $F(x) < G(x)$  for all  $x \in E^*$ . Three operators on these state variables make it possible to define state variables for large scale state systems while leaving  $F$  and often  $E$  implicit. The first is a kind of “zero”. If  $nulls$  is the empty sequence and  $\underline{y} = F(x)$  then **initial**  $\underline{y} = F(nulls)$ . So **initial**  $\underline{y}$  is the initial state value of  $\underline{y}$ . The second modifier is like “add one” - it extends the sequence by one event so that **after**  $e$   $\underline{y} = F(x.e)$  where  $s.e$  is

the sequence obtained by appending event  $e$  to sequence  $s$ . The third operator is for substitution and, as shown below, can be used to define state variables of systems that are constructed by interconnecting parallel components in layers. For the same  $y$  let **sub**  $u$   $y = F(u)$ . The fun cases are when  $u$  is itself a state variable. Suppose  $z$  is sequence valued in which case **sub**  $z$   $y = F(z) = F(G(x))$ .

The approach outlined above is built on three components. The first is a conception of state machines (automata) as maps  $F : E^* \rightarrow X$  that was mentioned in a number of works from the 1960s but made explicit in for example Arbib [1] in a slightly different form. Second, Peter [7] defined a class of primitive recursive word functions that extended primitive recursion on integers to primitive recursion on words (sequences). This formulation inspires the first two operators - although, it is related to Arbib's sequentiality property. The last operator imports a general automata product [3] — perhaps the biggest jump here. The general product offers a way of arbitrarily connecting state machines so that outputs can help steer inputs. This feedback of outputs is essential for representing how computer components connect but mainstream automata theory was too entranced by Krohn-Rhodes[5] theory to look into it much.

The next section clarifies the state machine and primitive recursive function basis of the methods. Section 3 illustrates applications to simple examples and to reliable message delivery. There is a short final section on more compact notation – the limited notation used here is wordier than it needs to be in an effort to make it more readily comprehensible.

## 2 State machines

Classically, if a map  $\gamma$  is defined by a pair of equations  $\gamma(0) = x_0$  and  $\gamma(n+1) = \alpha(\gamma(n))$  where  $\alpha$  is already defined, we know that  $\gamma$  is well defined on all non-negative integers. This construction is called “primitive recursion”. Peter[8] observed that the same construction works for finite sequences where appending takes the place of “add one”. So if  $F(\text{nulls}) = x_0$  and  $F(w.e) = G(F(w), e)$  then, if  $G$  is well-defined, so is  $F$ . This construction is fundamental to how state machines operate: computing the “next state” from the “current state” and the next input event.

### 2.1 State machines

A deterministic state machine is usually given with some variant of:  $M = (E, X, S, \iota, \delta, \lambda)$  where set  $E$  is the alphabet of events, set  $X$  is the set of outputs. The set  $S$  is the state set and  $\iota \in S$  is the initial state. The map  $\delta : S \times E \rightarrow S$  is the transition map and  $\lambda : S \rightarrow X$  is the output map which defines the difference between internal state and visible state. I am going to call this a “generalized Moore machine” because it is a classical Moore machine[6] without a restriction to finite sets. All actual digital computing devices are finite state, but infinite state machines can be useful abstractions even when thinking about finite devices - as seen in the examples below.

Given a generalized Moore machine  $M = (E, X, S, \iota, \delta, \lambda)$  the primitive recursive extension of  $\delta$  to sequences in  $E^*$  is straightforward. Let  $\delta^*(null) = \iota$  and let  $\delta^*(w.e) = \delta(\delta^*(w), e)$ . To avoid wasting another letter define:  $M^*(w) = \lambda(\delta^*(w))$ . The map  $M^*$  is a primitive recursive function on finite words over  $E$ .

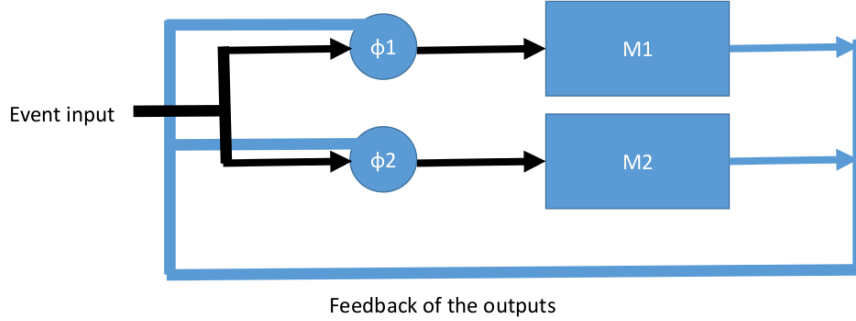
The construction of a generalized Moore machine from a map  $f : E^* \rightarrow X$  is straightforward as well. Suppose  $f : E^* \rightarrow X$  is given and define an equivalence relation as follows:

$$\text{if } w, z \in E^* \text{ then } w \sim_f z \text{ iff } (\forall u \in E^*) f(\text{concat}(w, u)) = f(\text{concat}(z, u))$$

where *concat* is concatenation of sequences. Note that if  $w \sim_f z$  then  $w.e \sim_f z.e$ . Partition  $E^*$  into sets  $\{w\}_f$  where  $\{w\}_f = \{z : z \in E^*, w \sim_f z\}$ . Let  $S_f = \{\{w\}_f : w \in E^*\}$  be the state set<sup>1</sup>. Then define  $\delta_f(\{w\}_f, e) = \{w.e\}_f$  and  $\lambda_f(\{w\}_f) = f(w)$ . The state machine defined in this way  $M_f$  has initial state  $\{null\}_f$ . It's easy to see that  $M_f^* = f$ . This type of construction is basically the same as state machine minimization and is well known in a slightly different context<sup>2</sup>.

## 2.2 State machine products

Gecseg [3] has a nice formulation of a “general product” of automata that is fairly simple as a block diagram.



Adapting it to generalized Moore machines, the idea is that we have an alphabet of events  $E$ , a collection of (not necessarily distinct) generalized Moore machines  $M_1 \dots, M_n$  and a collection of maps  $\phi_i : E \times X_1 \times \dots \times X_n \rightarrow E_i$ . The product machine is constructed by letting each  $\phi_i$  steer each component state machine depending on the input from  $E$  and the “feedback” which is the output of the component machines.

<sup>1</sup>If we define the congruence by  $z \cong_f w \leftrightarrow (\forall u, v)(f(uzv) = f(uwz))$  then instead of a state set we get a monoid under concatenation of representative elements.

<sup>2</sup> There are an infinite number of distinct generalized Moore machines that generate the same primitive recursive map  $M^*$ , but the differences between those machines are not interesting for our purposes. In fact, those differences are essentially artifacts of the presentation if all we are interested in is modeling the behavior of discrete state systems.

The product machine  $M = \Pi_{i=1}^n [M_i, \phi_i]$  has a state set consisting of the cross product  $S = \times_{i=1}^n S_i$  and a transition map  $\delta(s, a) = (\delta(s_1, e_1), \dots, \delta(s_n, e_n))$  where  $s = (s_1, \dots, s_n)$  and  $e_i = \phi_i(e, \lambda_1(s_1), \dots, \lambda_n(s_n))$ <sup>3</sup>.

The primitive recursive nature of the general product should be reasonably clear. If  $f_i = M_i^*$  then define  $g_i(nulls) = nulls$  and  $g_i(w.s) = g_i(w). \phi_i(e, g_1(w), \dots, g_n(w))$ . Then  $F(w) = (f_1(g_1(w)), \dots, f_n(g_n(w)))$  defines  $F$  so that  $F = M^*$  for the product machine  $M$ .

## 2.3 State variables

For any generalized Moore machine,  $M$ , then, there is a  $\underline{y}$  so that  $\underline{y} = M^*(x)$ . If  $\underline{y}$  is specified in terms of **initial** and **after**  $e$  then there is an  $M$  so that  $\underline{y} = M^*(x)$ . Of course it is possible to define state variables that do not have solutions or that have many solutions. To accomplish the product construction with state variables, assume that we have  $\underline{y}_1 \dots, \underline{y}_n$  where each  $\underline{y}_i = f_i(x)$ . Now construct  $\underline{u}_i$  by

$$\text{initial } \underline{u}_i = nulls \quad (1)$$

$$\text{after } e \underline{u}_i = \underline{u}_i. \phi_i(e, \text{sub } \underline{u}_1 \underline{y}_1, \dots, \text{sub } \underline{u}_n \underline{y}_n) \quad (2)$$

$$\underline{Y} = (\text{sub } \underline{u}_1 \underline{y}_1, \dots, \text{sub } \underline{u}_n \underline{y}_n) \quad (3)$$

A further step where components can move multiple steps on a single step of the product is also simple and preserves the state machine semantics. For this case the maps  $\phi_i$  are sequence valued and we use concatenation instead of appending.

$$\text{after } e \underline{u}_i = \text{concat}(\underline{u}_i, \phi_i(e, \text{sub } \underline{u}_1 \underline{y}_1, \dots, \text{sub } \underline{u}_n \underline{y}_n))$$

Say that a state variable  $\underline{y}$  is “finite state” if  $\underline{y} = f(x)$  where  $S_f$  (the partition of  $E^*$  by the equivalence relation  $\sim_f$ ) is finite. Note that if each  $\underline{y}_i$  is finite state, then  $\underline{Y}$  is finite state in the construction above even when the feedback maps are finite sequence valued.

## 3 Illustrative example

### 3.1 Simple examples

I’m going to define some examples  $\underline{y} = f(x)$  and then “solve for”  $f$  to build some intuition about what the operators do.

---

<sup>3</sup>This type of product was described in a by-the-way manner in multiple works in early automata research – for example in Hartmanis[4]. Researchers, however, early on became focused on “loop free” decomposition of automata for a number of reasons including the very interesting relationship between automata decomposition and group theory that is described in the Krohn-Rhodes theorem. By the 1970s, more general automata products were such an obscure topic that people writing papers about formal methods routinely claimed that it was not possible to model interaction and parallel computation with plain old state machines.

- A mod  $c$  counter that counts events:

$$\begin{aligned} \mathbf{initial} \ \underline{C} &= 0, \\ \mathbf{after} \ e \ \underline{C} &= \underline{C} + 1 \bmod c. \end{aligned}$$

Implicitly  $\underline{C} = f(x)$  for some  $f$ . Since  $\mathbf{initial} \ \underline{C} = f(\text{nulls}) = 0$  and

$$\begin{aligned} \mathbf{after} \ e \ \underline{C} &= f(x.e) \\ &= \underline{C} + 1 \bmod c \\ &= f(x) + 1 \bmod c \end{aligned}$$

$f$  is completely defined and so is  $\underline{C}$ .

- An unbounded counter that counts events:

$$\begin{aligned} \mathbf{initial} \ \underline{C_{unbounded}} &= 0, \\ \mathbf{after} \ e \ \underline{C_{unbounded}} &= \underline{C_{unbounded}} + 1. \end{aligned}$$

- Two connected mod  $c$  counters - to illustrate substitution. First define  $\underline{D}$  to be composed from the previously defined mod  $c$  counter  $\underline{C}$  and  $\mathbf{sub} \ \underline{u} \ \underline{C}$  where  $\underline{u}$  is to be defined. Here  $*$  is just ordinary multiplication.

$$\underline{D} = \underline{C} + c * \mathbf{sub} \ \underline{u} \ \underline{C}$$

Now define  $\underline{u}$ .

$$\begin{aligned} \mathbf{initial} \ \underline{u} &= \text{nulls}, \\ \mathbf{after} \ e \ \underline{u} &= \begin{cases} \underline{u}.e & \text{if } \underline{C} = c - 1; \\ \underline{u} & \text{otherwise.} \end{cases} \end{aligned}$$

Solve for  $g$  so that  $\underline{u} = g(x)$  remembering that  $\underline{C} = f(x)$ .

$$\begin{aligned} \mathbf{initial} \ \underline{u} &= g(\text{nulls}) = \text{nulls} \\ \mathbf{after} \ e \ \underline{u} &= g(x.e) \\ &= \begin{cases} \underline{u}.e = g(x).e & \text{if } f(x) = c - 1; \\ \underline{u} = g(x) & \text{otherwise.} \end{cases} \end{aligned}$$

so  $g$  is completely defined and so is  $\underline{u}$ . Now solve for  $h$  so that  $\underline{D} = h(x)$ .

$$\begin{aligned} h(x) &= \underline{D} \\ &= \underline{C} + c * \mathbf{sub} \ \underline{u} \ \underline{C} \\ &= f(x) + c * f(g(x)) \end{aligned}$$

- It is not necessary for state variables to be scalar valued. In fact, because of the nature of computer systems where code and data are fluid categories, map valued state variables are natural. Consider a queue to be a map from non-negative integers to queued values where  $\underline{Q}(1)$  is the first element,  $\underline{Q}(2)$  is the second and so on. To start, I'll assume we have no bound on the length of the queue, although such a thing is impossible in real-life,

and that there is some special element  $nullv$  so that  $Q(i) = nullv$  if there is no value at position  $i$ . Suppose there is a set  $V$  of possible values to be put in the queue (of course,  $nullv \notin V$ ). The events that change queue state are  $\mathbf{deq}$  and  $\mathbf{enq}_{[v]}$  for  $v \in V$ . Events not in those sets are just ignored.

$$\begin{aligned}
& \underline{Q}(i) = nullv \text{ and } j > i \rightarrow \underline{Q}(j) = nullv \\
& \mathbf{initial} \ \underline{Q}(1) = nullv \\
& \mathbf{after} \ e \ \underline{Q}(i) = \begin{cases} v & \text{if } e = \mathbf{enq}_{[v]}, v \in V \text{ and } i = 1 \\ \underline{Q}(i-1) & \text{if } e = \mathbf{enq}_{[v]}, v \in V \text{ and } i > 1 \\ \underline{Q}(i+1) & \text{if } e = \mathbf{deq} \\ \underline{Q}(i) & \text{otherwise} \end{cases}
\end{aligned}$$

- What about a bounded length queue? There are a number of alternatives for what happens when someone tries to push such a queue beyond its boundaries. One is to just leave it unspecified. Suppose the queue has a maximum length of  $c$  elements. Then we might only specify behavior when the queue is not empty and has never been over-filled.

$$\begin{aligned}
& \underline{CQ} \text{ is a } c \text{ queue if } \underline{Q}(1) \neq nullv \text{ and } \underline{HighWater} < c \rightarrow \underline{CQ} = \underline{Q}(1) \\
& \mathbf{initial} \ \underline{HighWater} = 0 \\
& \mathbf{after} \ e \ \underline{HighWater} = \max\{\underline{HighWater}, \max\{i : Q(i) \neq nullv\}\}
\end{aligned}$$

In this case,  $\underline{Q}$  is just used to define how the actual bounded queue must behave. There are an infinite number of maps  $h$  so that  $\underline{CQ} = h(x)$  since nothing is said about the value of  $\underline{CQ}$  if the high water mark is ever passed or if the queue is empty<sup>4</sup>.

### 3.2 Reliable broadcast

For a less trivial example, I'm going to describe a computer network and algorithms for committing messages. A common set of network properties is that message transmit is a broadcast (can be received by some or all other nodes), is unreliable (in that messages may not be received), and that there are no spurious messages (a received message must have been sent). To "commit" a message, a sender has to be assured that all recipients in a group have received the message. Describing this system in terms of state variables permits us to leave most of the behavior of the network unspecified, which is good because the network has an enormous number of states and depends on many parameters we don't want to even think about here.

I want to begin with state variables for network nodes (the computers connected to the network). These are undoubtedly complex objects, but we only

<sup>4</sup>The so-called "formal methods" literature is replete with many examples of researchers mistaking "not-specified" for "non-deterministic". The first is a quality of the specification. The second is a property of a mathematical object. Specifying that an engineered computer system is non-deterministic, seems to me something that one would rarely if ever need to do.

need some simple properties. Suppose we have a set *Messages* of messages. and a state variable  $\underline{T}$  that indicates what, if any, message the node wants to transmit in the current state. As is common with variables like this, we need a value  $nullm \notin Messages$  to indicate "no message", so  $\underline{T} = m \in Messages$  means that in the current state the node is attempting to transmit  $m$  and  $\underline{T} = nullm$  means the node has nothing to say in the current state. The event alphabet  $E_{node}$  should include one event  $\mathbf{tx}$  to indicate that a message has been transmitted and one event  $\mathbf{rx}[m]$  for each  $m \in Messages$  to indicate that message  $m$  has been received by the node <sup>5</sup>. There can be any number of other events in  $E_{node}$  but those are not things we have to specify here.

To keep track of which messages the node has received and sent we can define two state variables that are set valued.  $\underline{R}$  is the set of messages that have been received and  $\underline{S}$  is the set of messages that have been sent. These don't necessarily correspond with anything implemented in the node computer, they are just abstract properties:

$$\begin{aligned} \mathbf{initial} \ \underline{R} &= \mathbf{initial} \ \underline{S} = \emptyset \\ \mathbf{after} \ e \ \underline{R} &= \begin{cases} \underline{R} \cup \{m\} & \text{if } e = \mathbf{rx}[m] \text{ for } m \in Messages; \\ \underline{R} & \text{otherwise.} \end{cases} \\ \mathbf{after} \ e \ \underline{S} &= \begin{cases} \underline{S} \cup \{\underline{T}\} & \text{if } e = \mathbf{tx} \text{ and } \underline{T} \in Messages; \\ \underline{S} & \text{otherwise.} \end{cases} \end{aligned}$$

In a more detailed treatment, we'd probably want to make messages expire so that that, e.g. at some point a message  $m$  is no longer considered to be "received" by a node. Expiry would allow reuse of messages. For this example, however, we can assume the set of messages is effectively infinite.

Suppose we have a set  $N$  of node names and want to connect nodes together. There will be an alphabet  $E$  for the network and a relationship between network events and node events. For each node  $n$ , there is an event sequence  $\underline{u}_n \in E_{node}^*$ . Then  $\mathbf{sub} \ \underline{u}_n \ \underline{T}$  is the state variable  $\underline{T}$  in the context of the state of node  $n$ .

Unlike the simple examples above, we don't know nearly enough to specify these completely. The initial state value will just have the node name – so that each node begins life knowing its own name.

$$\mathbf{initial} \ \underline{u}_n = nulls.n$$

We can then require that each node know its own id:

$$\mathbf{sub} \ \underline{u}_n \ \underline{Id} = n$$

Node internal events may be very complex, but all we care about is interaction with the network. First, to keep it simple, make sure that when the network advances one step, each node advances at most one step.

$$\mathbf{after} \ e \ \underline{u}_n \in \{\underline{u}_n, \underline{u}_n.e', e' \in E_{node}\}$$

---

<sup>5</sup> The symbols  $\mathbf{rx}[m]$  are, again, just indexed symbols. Think of the  $[m]$  as subscripts that are not small font (easier to read) and higher than normal if you want.

A  $\text{rx}[m]$  event needs to be associated with a  $\text{tx}$  event of the same message  $m$ <sup>6</sup>.

$$\text{after } e \text{ } \underline{u}_n = \underline{u}_n.\text{rx}[m] \rightarrow (\exists n')(\text{after } e \text{ } \underline{u}_{n'} = \underline{u}_{n'}.\text{tx} \\ \text{and } \text{sub } \underline{u}_{n'} \text{ } \underline{T} = m \in \text{Messages})$$

There is no reciprocity - a  $\text{tx}$  event may not be associated with any receive at all. These constraints model an unreliable broadcast network with no spurious messages.

**Lemma 1**  $m \in \text{sub } \underline{u}_{n_1} \text{ } \underline{R} \rightarrow (\exists n_2)(m \in \text{sub } \underline{u}_{n_2} \text{ } \underline{S})$

Proof by induction on state. **initial**  $\text{sub } \underline{u}_{n_1} = \emptyset$  so there is nothing to prove. Assuming the lemma in the current state, there is only one interesting case  $m \notin \text{sub } \underline{u}_{n_1} \text{ } \underline{R}$  and  $m \in \text{after } e \text{ } \text{sub } \underline{u}_{n_1} \text{ } \underline{R}$ . In this case  $\text{after } e \text{ } \underline{u}_{n_1} = \underline{u}_{n_1}.\text{rx}[m]$  so  $(\exists n_2)\text{sub } \underline{u}_{n_2} \text{ } \underline{T} = m$  and  $\text{after } e \text{ } \underline{u}_{n_2} = \underline{u}_{n_2}.\text{tx}$

The challenge is for one node to determine whether a group of other nodes have received a particular message. That is, can we define a boolean map valued state variable Commit so that for some  $G \subset N$ :

$$\text{sub } \underline{u}_n \text{ } \underline{\text{Commit}}(G, m) = 1 \rightarrow (\forall n' \in G)m \in \text{sub } \underline{u}_{n'} \text{ } \underline{R}$$

The obvious solution is to make each recipient send a matching acknowledgment message for each data message. Suppose we have a subset  $D \subset \text{Messages}$  so that for each  $d \in D$  and each  $n \in N$  there is a unique  $a_{d,n} \in \text{Messages}$  which is an acknowledgment of  $d$  from  $n$ . Require that a site transmit  $a_{d,n}$  only if it has received  $d$  and either it is node  $n$  or it is resending an ack it already received:

$$\underline{T} = a_{d,n} \rightarrow d \in \underline{R} \text{ and } (\underline{Id} = n \text{ or } a_{d,n} \in \underline{R}).$$

**Lemma 2**  $a_{d,n'} \in \text{sub } \underline{u}_n \text{ } \underline{R} \implies d \in \text{sub } \underline{u}_{n'} \text{ } \underline{R}$ .

Let's prove this for  $(\exists n)a_{d,n'} \in \text{sub } \underline{u}_n \text{ } \underline{R} \implies a_{d,n'} \in \text{sub } \underline{u}_{n'} \text{ } \underline{S}$  using induction on state. Clearly true in the initial state. Suppose the left side is false in the current state and true after  $e$ . Then by lemma 1 there is some  $n''$  so that after  $e$   $a_{d,n'} \in \text{sub } \underline{u}_{n''} \text{ } \underline{S}$ . The hypothesis says that  $a_{d,n'} \notin \text{sub } \underline{u}_{n'} \text{ } \underline{R}$  so  $n' = n''$  and  $d \in \text{sub } \underline{u}_{n'} \text{ } \underline{R}$ . Once a message is in  $\underline{R}$  it stays there, so QED.

If  $\underline{\text{Commit}}(G, m)$  is defined to be  $(\forall n \in G)(a_{d,n} \in \underline{R})$  the main result follows immediately.

A better method relies on cycles of receivers each taking a turn [2] to broadcast an ack. Suppose  $|G| = k$  is the number of nodes in the group and  $\pi : \{0 \dots k-1\} \rightarrow G$  is onto. Further suppose that data and ack messages carry sequence numbers so  $d_i$  and  $a_i$  have the same sequence number  $i$ . First, we need to require that sequence numbers of messages that are sent are uniquely identifying (a more detailed treatment would cycle sequence numbers).

$$d_i \in \text{sub } \underline{u}_{n_1} \text{ } \underline{S} \text{ and } d'_i \in \text{sub } \underline{u}_{n_2} \text{ } \underline{S} \rightarrow d'_i = d_i \quad (4)$$

<sup>6</sup>Here I am providing a really simple model of a network where messages pass directly between nodes. A more realistic model would have a network medium of some sort.



Next: a node can send an ack  $a_i$  only if is resending it (if it received it already) or if  $\pi$  maps the sequence number mod  $k$  to the node identifier. As we increase sequence numbers, we cycle through the set of nodes.

$$\underline{T} = a_i \rightarrow \pi(i \bmod k) = \underline{Id} \text{ or } a_i \in \underline{R} \quad (5)$$

Finally, the core of the algorithm is that acks are only sent if all previous messages and acks have been received.

$$\underline{T} = a_i \rightarrow \begin{cases} \exists d_i \in \underline{R} \text{ and } \forall j < i, \exists a_j, d_j \\ a_j \in \underline{R} \text{ and } d_j \in \underline{R} \end{cases} \quad (6)$$

Suppose  $d_i \in \mathbf{sub} \ \underline{u}_n \ R$  and Suppose  $a_{i+k} \in \mathbf{sub} \ \underline{u}_n \ \underline{R}$ . It follows that  $a_{i+k} \in \mathbf{sub} \ \underline{u}_{\pi((i+k) \bmod k)} \ \underline{S}$  which implies that there is a complete cycle of nodes in  $G$  that have received  $d_i$ .

## 4 Some notes

### 4.1 Simplified notation

The goal here has been something that would help for sketching out designs on paper – and perhaps a basis for some automated tools as well. For scratching out specifications on paper, however, the notation for the three operators is a little wordy. Usually, I omit the underline when the state variables are clear from context. In place of **initial**  $y$ , I often write  $\downarrow y$ . In place of **after**  $e \ y$  we can write  $ey$  – just juxtaposition. And **sub**  $u \ y$  can be written  $u|y$  borrowing some similar semantics from UNIX[9] where  $|$  is used to connect the output of the value on the left to the input of the value on the right. The general product equations starting at equation 1 above would then look like the following.

$$\downarrow u_i = \text{nulls}$$

$$eu_i = u_i.\phi_i(e, u_1|y_1, \dots u_n|y_n)$$

$$Y = (u_1|y_1, \dots u_n|y_n)$$

Notation is not the point of this work<sup>7</sup> but using this notation declutters things enough to better see, for example, what loop-free products look like.

### 4.2 Loop free and nearly loop free products

In a loop-free or cascade product  $\phi_i$  references only  $u_i|y_i \dots u_n|y_n$ :

$$eu_i = u_i.\phi_i(e, u_i|y_i, \dots u_n|y_n).$$

In this case, there is no feedback - each component  $i$  sees only the output from the higher numbered components. There is a lot of work in algebraic automata

---

<sup>7</sup>This is not a formal methods paper.

theory on the algebraic consequences of such a design[5]. Systems that can be designed with this kind of communication pattern are perhaps amenable to certain optimizations. I wonder also if decompositions that are “almost” loop-free have interesting properties. Consider a shift register that shifts only left. Let  $\underline{cell}$  be a state variable for a single storage cell:  $e\underline{cell} = e$ . A register of  $n$  cells looks like  $\underline{Register} = (u_1|\underline{cell}, \dots u_n|\underline{cell})$ . A loop free shift right register is implemented by defining  $u_i$  as follows:

$$\begin{aligned} \downarrow u_i &= \text{nulls} \\ eu_i &= \begin{cases} u_i.(u_{i-1}|\underline{cell}) & \text{if } i > 0; \\ u_i.e & \text{if } i = 1 \end{cases} \end{aligned}$$

Suppose, though that elements of  $E$  are  $(1, v)$  (shift in  $v$  to the right) and  $(-1, v)$  (shift in  $v$  to the left).

$$(r, v)u_i = \begin{cases} u_i.(u_{i-r}|\underline{cell}) & \text{if } 0 < r - i < n; \\ u_i.v & \text{otherwise} \end{cases}$$

In this case,  $\underline{Register}$  is definitely not loop-free, but it operates in a loop-free way for each event. Does that mean anything either algebraically for the underlying monoid or for the structure of a primitive recursive word function? Certainly, engineers work to reduce communication and synchronization complexity in system design – e.g. by using time synchronization to replace handshake protocols.

## References

- [1] M. A. Arbib. *Algebraic theory of machines, languages, and semi-groups*. Academic Press, 1968.
- [2] J.-M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.
- [3] F. Gecseg. *Products of Automata*. Monographs in Theoretical Computer Science. Springer Verlag, 1986.
- [4] J. Hartmanis. Loop-free structure of sequential machines. In E. Moore, editor, *Sequential Machines: Selected Papers*, pages 115–156. Addison-Welsey, Reading MA, 1964.
- [5] W. Holcombe. *Algebraic Automata Theory*. Cambridge University Press, 1983.
- [6] E. Moore, editor. *Sequential Machines: Selected Papers*. Addison-Welsey, Reading MA, 1964.
- [7] R. Peter. *Recursive functions*. Academic Press, 1967.

- [8] R. Peter. *Recursive Functions in Computer Theory*. Ellis Horwood Series in Computers and Their Applications, 1982.
- [9] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, July 1974.